NPS-CS-03-003
February 2003

The Center for INFOSEC Studies and Research

white paper

Execution Policies Research and Implementation

Paul C. Clark, Timothy E. Levin, Cynthia E. Irvine

Center for Information Systems Security Studies and Research
Computer Science Department
Naval Postgraduate School
Monterey, California 93943

Execution Policies

Research and Implementation

Paul C. Clark, Timothy E. Levin, Cynthia E. Irvine

Center for Information Systems Security Studies and Research
Computer Science Department
Naval Postgraduate School
Monterey, California 93943

Abstract:

This research studied the application of a software-based ring execution policy, the type of which has previously been implemented via hardware mechanisms, to an open source operating system. Such an execution policy is orthogonal to, and may be used in conjunction with, other mandatory (viz, secrecy, integrity) and discretionary policies. It allows processes running with otherwise similar privileges (such as the root user, or secrecy attributes) to be differentiated with respect to priority or privilege regarding system resources and execution. We have found that it is possible to construct a mandatory ring execution policy whose primary function is to restrict subjects from executing certain file system objects, and that this may result in a more coherent and manageable policy than what can be expected from various discretionary (e.g., policy-bypass or privilege-grouping) mechanisms.

Background

This research studied the application of a software-based ring execution policy, the type of which has previously been implemented via hardware mechanisms, to an open source operating system. A mandatory access control (MAC) ring execution policy restricts execution of programs based on labels associated with both the programs to be executed, and the process requesting the execution (such as a shell). In this approach, each program executes in a specific domain, or ring. A ring number stored in the process's MAC security label determines this domain. The range of domains from which an object may be executed is stored in the object's security label. By comparing the label of the process to the label of the file to be executed, the kernel can thus make a determination whether the action was permitted, based on the execution policy.

In contrast to a MAC policy, the normal Unix access control policies are discretionary (DAC), allowing the permissions on a file to be given/denied as seen fit by the owner. Similarly, various *privilege-grouping* mechanisms allow for ad hoc organizations of privileges to bypass the nominal security mechanisms. These policies are very susceptible to human error and malicious code, such as Trojan horses. Furthermore, they are not persistent, leading to a situation in which the security state of the system might be unknown at any given time with respect to the "execution" policy.

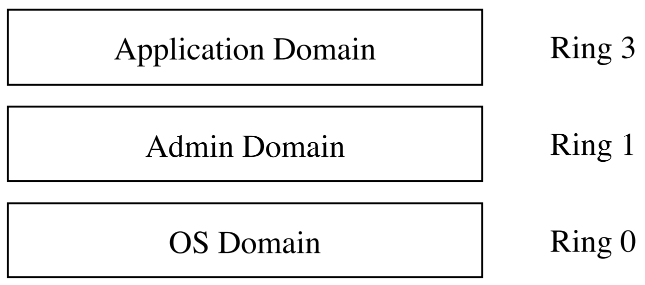
Our initial research concentrated on developing a meaningful access control policy based on rings. The original abstraction for the ring policy was based on the hardware privilege mechanism of the Intel x86 processors, dating back to Multics [SCHR72], where four execution rings (or domains) are supported. The resulting policy was a hybrid of a policy described by Gasser in *Building a Secure Computer System* [GAS95-1], and a policy used by the high assurance Gemini kernel [FAI95].

A Third Execution Domain

Presently, Unix systems provide two execution domains: one for "user" programs and one for the kernel or operating system (OS). It is proposed that a new domain be added between the existing OS and user domains. The potential uses of this new domain are:

- Execution of "trusted applications" in a more controlled environment.
- Improved integrity of the system by limiting what trusted applications can read.
- Restricted access to security-relevant system calls to "trusted applications".
- More protection for file system objects that are used to configure the system.

This new domain is therefore dubbed the "Admin Domain", and would reside in ring 1 (using the Multics/Intel vocabulary).



Execution Policy and Mechanism

The development of the execution policy begins as follows. Given that processes execute in an assigned execution ring, and that program files are assigned a range of rings from which they may be executed, a process can execute a program file if and only if the assigned ring of the process is within the range assigned to the program file (inclusive).

This is fairly straightforward; each file is assigned two numbers, indicating the range of processes that can execute it. These ring values are referred to as ring bracket 1 (RB1) and ring bracket 2 (RB2). A process can execute a file if its assigned execution ring meets the following criteria: RB1 <= (process execution ring) <= RB2 (with the requirement that RB1 <= RB2). Unfortunately, this simple starting point is insufficient.

To illustrate the shortcoming of the policy stated above, assume a simple environment where this is the only policy being enforced, and assume that newly created files are assigned both RB1 and RB2 ring values equal to the ring of process that creates it. If a process is denied execution permission on a file, it could get around the denial by copying the restricted executable, whereupon its ring values would change to something that would allow it to be executed.

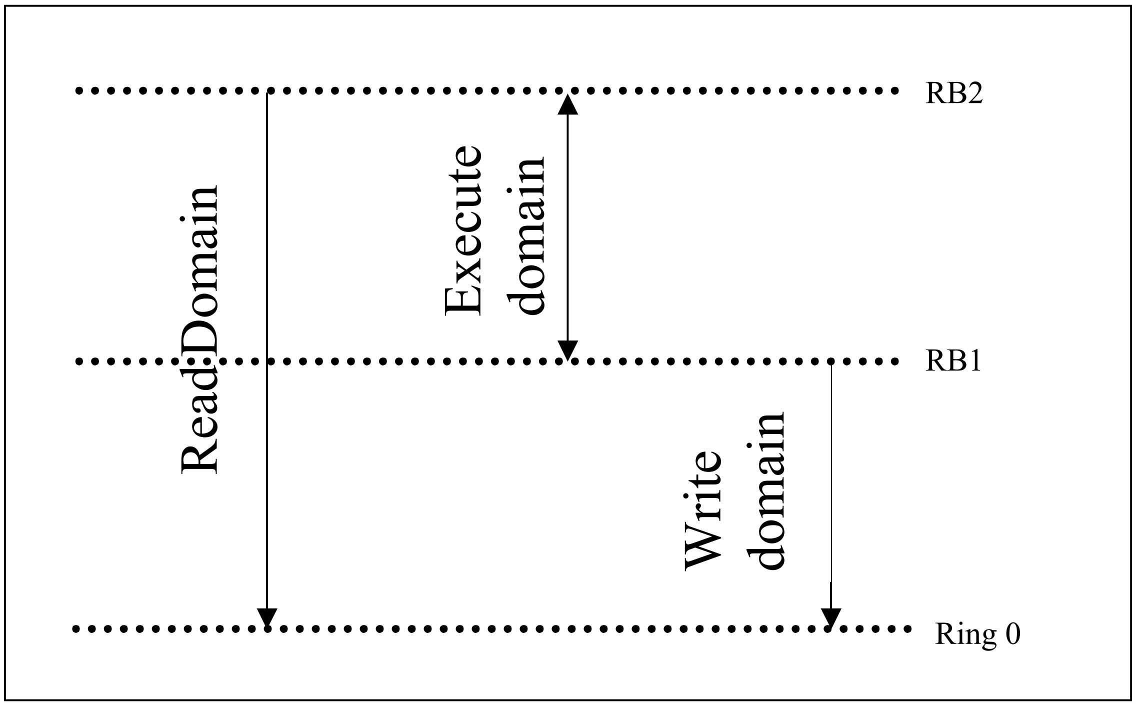
To close this hole, a read restriction must be enforced, as follows: a process can read an object if its execution ring is less than or equal to RB2. In other words, if a process is not allowed to execute a file, it is not allowed to read the file.

Unfortunately, there is another hole that must be plugged. Consider that a process or file ring number is indicative of a level of "privilege", which is inversely proportional to the ring number. On one hand, the execution policy can be used as a way to restrict less-privileged processes (viz., where the process ring is greater than RB2) from executing more-privileged files . On the other hand, it can also be used as a way of preventing more-privileged processes (viz., where the process ring is less than RB1) from executing less-privileged (lower integrity) files. With respect to the latter use, files must be write-protected from the less privileged (lower integrity) processes. This can be done as follows: a process can write an object if its execution ring is less than or equal to RB1. It can be argued that using RB2 would have had the same affect, but the more restrictive option prevents processes at RB1 from executing files modified by processes at RB2. Thus, it is assured that the integrity of a file is at least that of its RB1.

A complete description of the resulting policy is as follows:

- Every process is assigned an execution ring.
- Every object is assigned two ring values, RB1 and RB2, where
  0 <= RB1 <= RB2.
- A process can execute an object iff:
  RB1 <= (process ring) <= RB2.
- A process can write to an object iff:
  (process ring) <= RB1.
- A process can read an object iff:
  (process ring <= RB2).

The policy can also be shown visually, as seen in the following figure:



Access Domains Defined by Object Ring Brackets

Implementation

The proposed execution policy was implemented on an open source Unix system, OpenBSD version 3.1. The ring labels were stored in a fairly new OpenBSD feature known as Extended File Attributes, which allows arbitrary data to be associated with individual files.

The implementation of the actual policy logic was trivial. The difficulty revolved around the following:

- It was difficult to modify an existing operating system to handle something it was not designed to do. For example, The Extended File Attribute feature was written with the assumption that it was always called from user space, not the system space. Because the kernel needed to query the labels on objects being accessed, it had to be done in the system space. This required existing code to be copied to new functions, which were identical except for one line here and there. The workaround was trivial, but only after spending much effort to track down the problem.
- The Extended File Attribute implementation in OpenBSD version 3.1 was beta code at best. There didn't appear to be a coherent design description or abstraction layer.
- The internal locking protocol on kernel objects has gotten out of hand over the years, which led to additional work to ensure that proper cleanup was performed at

the end of a system call. One man page states: "the locking discipline is bizarre." [MAN]

Despite the difficulties, a working implementation was produced, though incomplete in its enforcement. The current implementation supports four rings, ranging from 0 to 3, but this number is fairly arbitrary. The number of effective rings can be any number that makes sense for a particular solution. In addition, there is currently no restriction to executing system calls based on the ring of the calling process.

The implementation supports the ability to configure the "clearance" of the system users. When adding a user to the system, a new configuration file must be updated by the system administrator to define which rings a user can request for the processes he creates. It is assumed that regular users would be restricted to the least privileged ring (3), while system administrators would have access to the inner rings. With this implementation, it is potentially possible to configure additional users with a UID of zero(normally associated with "root"), but with different user names, which would be restricted to the lesser privileged rings, resulting in the enforcement of the Principle of Least Privilege.

Configuration

Determining the proper ring attributes for specific file system objects is non-trivial. Indeed, it turns out that some simple questions must first be answered before delving into such a task: what is the primary purpose for using the ring policy? Will it be used to provide separation of duties? Will it be used to provide integrity of critical data files? Will it be used to provide integrity of critical programs? Are the executable files initially installed on the system "trusted" to be non-malicious?

Initial Configuration

The first configuration of the system was based on the desire to separate administrative functions from the user space, and to protect critical system configuration files from regular users. A by-product of this desire to protect critical system files was to "distrust" as many programs as possible, to prevent malicious activity from harming the system.

In the execution ring abstraction, the kernel executes in the most privileged ring (0). Trusted programs, or programs that need to be limited to administrators, were assigned to ring 1, while typical user programs were assigned to ring 3. Ring 2 was not used. Programs that are only supposed to be executed by ring 1 processes were labeled as RB1=RB2=1. [For what follows, notation will refer to ring settings as (RB1, RB2), or (1, 1) in this case.]. Programs that are to be executed by both ring 1 or 3 processes were labeled as (1, 3), while programs that are executed only by ring 3 processes were labeled as (3, 3). For example, the highly privileged initial Unix process (/sbin/init) is started by the kernel in ring 1, and was therefore labeled as (1, 1).

Init starts the initial terminal processes that prompt for a user's login name. This terminal process is executed from the file /usr/libexec/getty, which is labeled as (1, 1), so that init can execute it. getty processes also run in ring 1.

After a user enters a login name, the login program is executed to prompt for and authenticate the user password. The login program at /usr/bin/login is labeled as (1, 1) so that getty can execute it. Login also runs in ring 1. Related to login, the files in /usr/libexec/auth/ were set to (1, 1), because they are used during login.

The login program was modified to prompt for the desired execution domain after a correct password is entered. Login then creates the user's environment and executes a shell. The shell program at /bin/csh is labeled as (1, 3) so that login can execute it in ring 1, or a user can execute it in ring 3. This labeling of the shell was necessary for ring 1 users to be able to log in, but the shell is a large program. Should it be trusted? The only other options are: 1) use some other shell that might be more acceptable, e.g., smaller and less functional, or 2) write our own. Indeed, the entire ring 1 user interface could be reduced to a simplified menu interface with all the desired administrative options built in, but that is outside the scope of this research.

This minimal labeling gets a user logged in at ring 1 or 3. At this point, other decisions had to be made (as outlined below), though it was not an exhaustive study. Rather, decisions were made as problems occurred.

1. What is the minimal set of programs allowed to be executed only in ring 1, and whose ring brackets should be (1, 1)?

   Initially this turned out to be a short list. Besides those described above, the only other programs in this list are: chlabel and polctl. Both of these additional programs were written in support of this research. The chlabel program provides the user interface for setting the ring values on file system objects. The polctl program enables/disables enforcement of the ring policy.

   However, any program that should only be run by the system administrator should theoretically only be executable by ring 1 processes. In practice, that might be difficult. One problem encountered dealt with the adduser command, which is used to add new users to the system. It happens to be a perl script, not a binary program. For adduser to work in ring 1, the perl program would have to have its ring brackets set at (1, 3), so it can also be used in user space. This seemed to violate the initial desire to distrust as many programs as possible. The question then became: should this system force administrators to add users "by hand", or should perl be "trusted"? Initially, perl was set as (3, 3), requiring users to be added by hand.

2. What is the minimal set of programs allowed to be executed by rings 1 and 3, and whose ring brackets should be (1, 3)?

   This is a longer list than the programs that are labeled as (1, 1). It turns out that there are a number of programs required for basic usage. Initially, the following were labeled (1, 3):

cat	mkdir	pwd	passwd
chmod	mv	sync	shutdown
csh	rm	chown	reboot
cp	rmdir	label	vi
ls	ps	more	

There are probably many more that can be added; this is a minimal set, in order to traverse and maintain the file system, and provide minimal system administration activities.

3. What is the minimal set of programs allowed to be executed only by ring 3, and whose ring brackets should be (3, 3)?

   This turns out to be all other executable files that do not fall under the (1, 1) or (1, 3) categories.

4. What set of data files should be labeled as (1, 1)?

   The /etc/inittab/ file (used to configure /sbin/init/), and the files used to store the extended attributes were labeled as (1, 1). Other good candidates for this attribute would be files used by other system daemons.

5. What set of data files should be labeled as (1, 3)?

   This category was applied to files that are modified by system administrators, while running in ring 1, and need to be readable by programs running in ring 3. Specifically, all the files in the /etc/ hierarchy were set to (1, 3), as well as the files in /security/, where new policy configuration files are stored.

All other files not covered in any of the above five categories were labeled as (3, 3), but this can change, depending on the use of the system. More work needs to be done in this area to ensure that the intended security is enforced properly, while not breaking any of the applications.

Secure Shell (SSH)

As a proof of concept, the secure shell program was adapted to both log in the remote user at a requested ring, and to allocate the secure shell executable components to specific rings. This exercise was very successful, and is summarized in [NGUYEN].

Process Throttling

Another possible feature is to be able to throttle back all processes in a given ring, given the appropriate command or circumstances. For example, given some "emergency state", the system could enforce a policy where only processes running in a more privileged ring can execute, while all other processes go to sleep, or to a very low priority queue. Support of this feature would require at least one new system call, as well as changes to the process scheduling logic.

Relation of Execution Policy to Traditional MAC Policy

As stated earlier, the primary distinction between a ring based execution policy and a discretionary policy is that the ring policy is both global and persistent. Once a process or file are created, their ring attributes do not change. The strictest requirements for a mandatory policy are mathematical in nature, as described below [BRINK]:

> The labels can form a partially ordered set, such that the comparison of set members satisfies the following mathematical conditions:
>
> - Reflexivity
> - Antisymmetry
> - Transitivity

The execution policy is based on a totally ordered set of ring numbers, which is a restricted form of a partially ordered set.

Another definition of MAC says that under “mandatory controls, the system assigns both subjects and objects special security attributes that cannot be changed on request as can discretionary access control attributes…” [GASS95-2] Under this definition, the execution policy is also mandatory, as opposed to discretionary. The system does not let regular users and owners of file system objects change the ring properties of an object.

Additionally, this execution policy provides some additional flexibility over traditional MAC secrecy and integrity policies that only provide read and write controls: the execution policy provides distinct read, write and execute control. Traditional MAC policies typically equate execute access with read access. An execution policy may be used in conjunction with traditional secrecy, integrity and discretionary policies to protect critical data. (See [GAS95] and[FAI95]).

Conclusions and Observations

It is possible to construct a mandatory execution policy whose primary function is to restrict subjects from executing certain file system objects. Such an execution policy is orthogonal to, and may be used in conjunction with, traditional secrecy, integrity and discretionary policies. It allows processes running with otherwise similar privileges (such as the root user, or secrecy attributes) to be differentiated with respect to which one has more priority or privilege with respect to system resources and execution. One might be inclined to attempt to provide a similar solution by carefully constructing a number of secrecy and integrity labels. However, the secrecy and integrity policies were not designed to solve the kinds of problems that the ring policy is designed to solve. Even if it was possible, the resulting system might be very difficult to understand or manage correctly.

We are working on a trusted path mechanism by which user-ring assignments can be reliably requested. In the future, we plan to perform an extensive testing to ensure correct system functionality under the defined ring assignments for OpenBSD system files and processes. Experiments to provide emergency-mode process throttling capabilities are also planned. Finally, we would like to pursue experiments and comparative analyses to objectively determine whether a mandatory ring execution policy results in a more coherent and manageable policy that what can be expected from various ad hoc and discretionary (e.g., policy-bypass or privilege-grouping) mechanisms.

Some open question are:

- Is the policy described sufficient and usable?
- Are there existing system calls that should be accessible from ring1 and not accessible to ring 3?
- Are there services that no longer work with the new policy?
- What are the appropriate ring labels for all installed files?

References

[BRINK]	Information Security: An Integrated Collection of Essays, “Concepts and Terminology For Computer Security”, Brinkley and Schell, http://www.acsac.org/secshelf/book001/book001.html
[FAI95]	Faigin, Daniel, et. al., *Final Evaluation Report Gemini Computer Incorporated Gemini Trusted Network Processor*, National Computer Security Center, 1995, pp. 110-111.
[GAS95-1]	Morrie Gasser, *Building a Secure Computer System*, New York: Van Nostrand Reinhold, 1988, pp. 107-111.
[GAS95-2]	Gasser, Morrie, *Building a Secure Computer System*, New York: Van Nostrand Reinhold, 1988, pg. 61.
[MAN]	OpenBSD version 3.1, man page for vn_lock().
[NGUYEN]	Thuy D. Nguyen, et al, Policy Enforced Remote Login, NPS Technical Report NPS-CS-03-004, February, 2003.
[SCHR72]	Schroeder, M. D., and Jerome H. Saltzer, "A Hardware Architecture for Implementing Protection Rings," Comm ACM, 15, 3, March, 1972.